# API ReST avec Spring Boot

Développement d'API ReST en Java avec le framework Spring et ses briques telles que : Spring Boot, Spring Data JPA, Spring Security

# Introduction

# Prérequis

- Introduction au Web
- Programmation Orientée Objet en Java

# Installation de Jetbrains IntelliJ IDEA et Jetbrains Datagrip

## Licence

IntelliJ et Datagrip sont des outils professionnels sous licence. Heureusement, Jetbrains permet aux étudiants de bénéficier de licences gratuites pour tous ses outils. Elles sont accessible via le Github Student Pack. Une fois le student pack en main, rdv ici pour créer un compte Jetbrains en utilisant son compte Github.

## Avec JetBrain Toolbox

Jetbrains Toolbox est un petit utilitaire très pratique qui permet d'installer et mettre à jours les logiciels JetBrains en un clic.

### Installer Jetbrains Toolbox

Téléchargez et installez Jetbrains Toolbox

### Installer IntelliJ

Exécutez JetBrains Toolbox puis faites un clic droit sur son icône dans la barre d'état système de Windows. Trouvez ensuite IntelliJ et Datagrip dans la liste des applications proposée. Cliquez sur "*Install*" et attendez la fin du téléchargement.

# Sans JetBrains ToolBox

Téléchargez IntelliJ via ce lien et datagrip via ce lien, puis exécutez l'installateur. Suivez ensuite les instructions d'installation.

# Installer une base de donnée MySQL

Laragon est un outil qui package plusieurs outils pour le développement web. Nous allons l'utiliser la base de donnée MySQL. Téléchargez Laragon via ce lien,puis exécutez l'installateur. Suivez ensuite les instructions d'installation.

## Configurer Laragon

Lancez Laragon puis faites *clic droit sur le fond > MySQL > Change root password*. Mettez un mot de passe et conservez le bien il servira à se connecter à la base de donnée. Lancez ensuite le serveur en faisant *clic droit sur le fond > MySQL > Start MySQL*

## Créer une base de donnée

Vous pouvez créer une base de donnée pour chacun de vos projet en faisant : *clic droit sur le fond > MySQL > Create Database*, puis donnez lui un nom.

# Installer une base de donnée Oracle

- Télécharger Oracle
- Extraire
- Exécuter et suivre le programme d'installation -> Attention bien se rappeller le mot de passe admin qu'on a miss
- Ouvrir SQL plus
- Se connecter le compte system et le mot de passe que vous avez setup dans l'installation
- Exécuter la commande suivante :

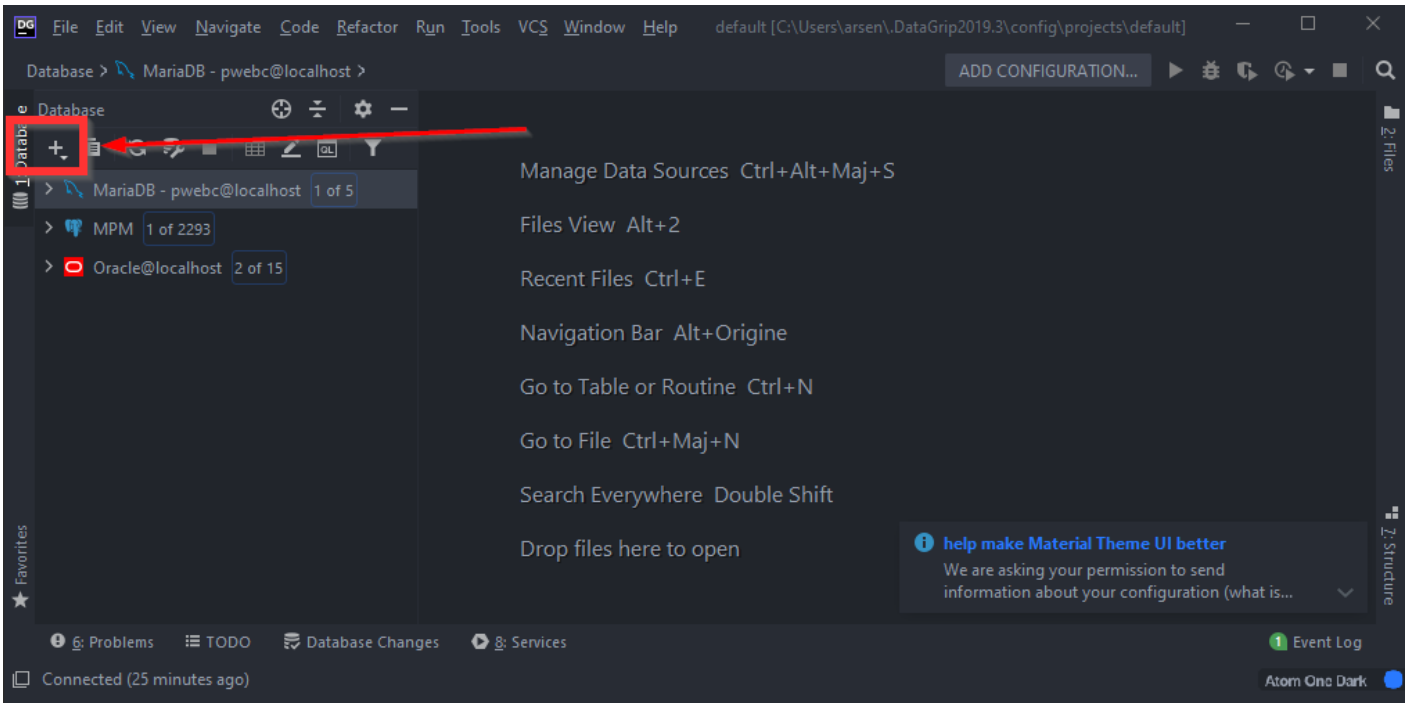
```
Exec DBMS_XDB.SETHTTPPORT(3010);
```

- Créer votre compte utilisateur :

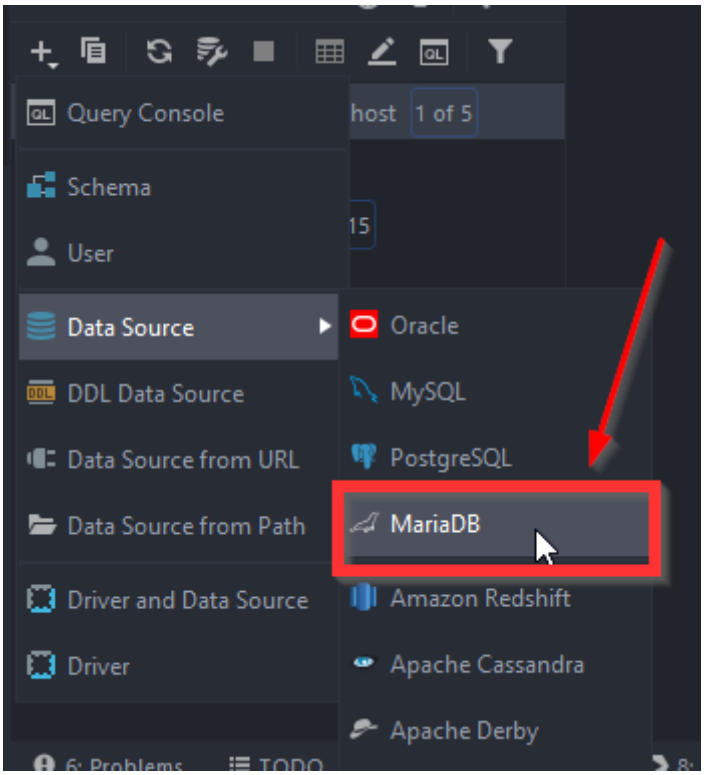
```
CREATE USER nom_utilisateur IDENTIFIED BY mot_de_passe;
GRANT DBA TO nom_utilisteur;
```

# Se connecter à une base de donnée avec Datagrip

Lancer Datagrip puis cliquer sur le bouton "+" :



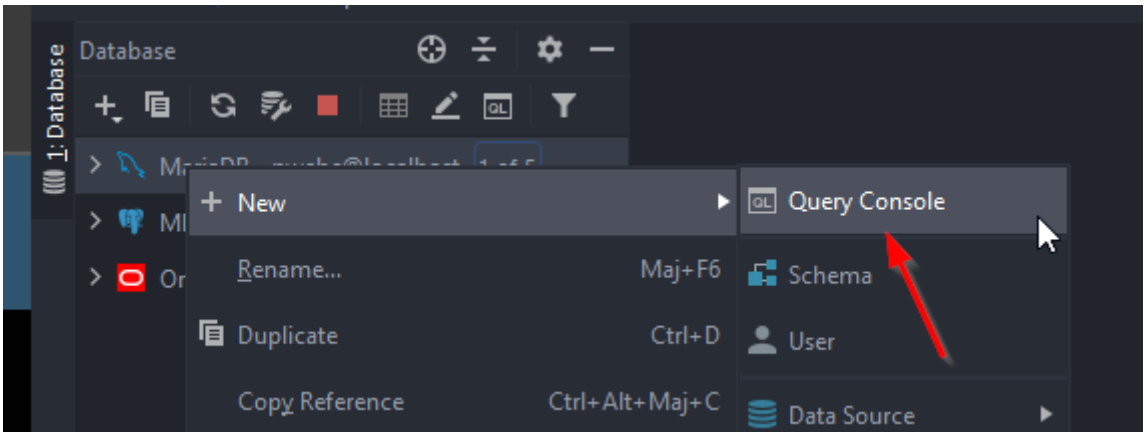

Et choisissez "*Datasource > MariaDB*" :




Au bas du formulaire cliquez sur "*Download missing driver file*". Remplissez ensuite le formulaire comme suit :

- Name : nom de votre projet
- Host : laissez `localhost`
- port : laissez `3306`
- User : `root`
- Password : Le mot de passe que vous avez configuré sur la base de donnée MySQL de Laragon
- Database : Le nom de la base de donnée que vous avez créé sur Laragon

Faites ensuite "*Test Connection*" afin de voir si tout est bien configuré, puis fait "*Apply*". Votre base de donnée est apparue dans la liste des bases de donnée à gauche de l'interface. Pour ouvrir une session SQL faites *Clic droit sur le nom de votre base* > *New* > *Query Console* :



Vous avez donc une console dans laquelle vous pouvez écrire des scripts SQL. `CTRL + ENTER` pour exécuter votre script.

# Installer un client HTTP

Pour interagir avec nos API ReST nous allons avoir besoin d'un client HTTP. Vous pouvez utiliser Postman, qui est téléchargeable ici.

# Le Framework Spring

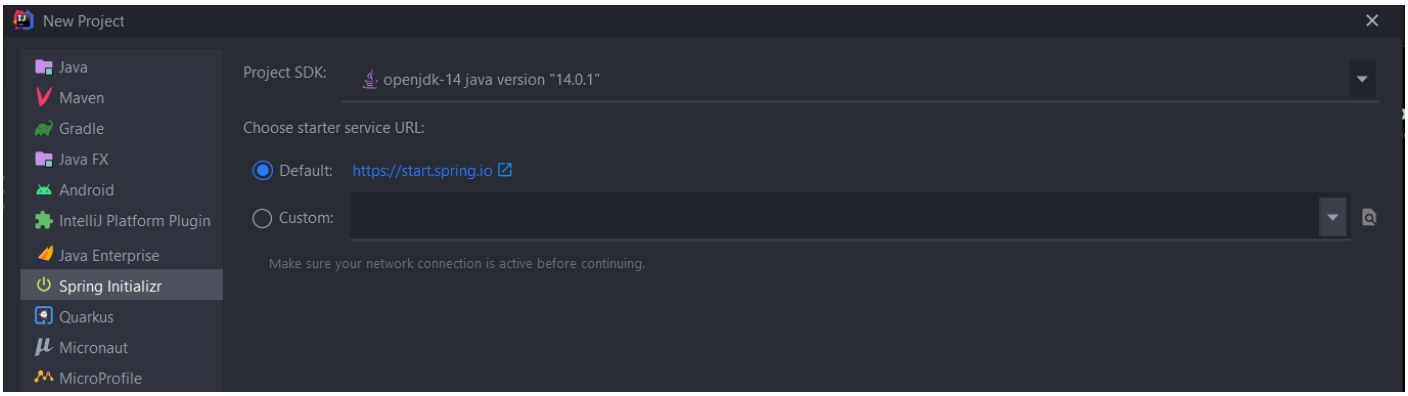
Spring est un Framework d'application Java open source qui est centré sur l'injection de dépendances. Il fourni également beaucoup de briques logicielles permettant de faciliter le développement d'application. On peut citer par exemple :

- Spring MVC pour les interractions Web
- Spring Data JPA pour se connecter à des bases de données
- Spring Security pour sécuriser les interactions avec le client
- Spring Boot pour l'autoconfiguration

Dans ce tutoriel nous allons apprendre à utiliser ces briques afin de développer des application web exposant des services sous la forme d'API ReST.

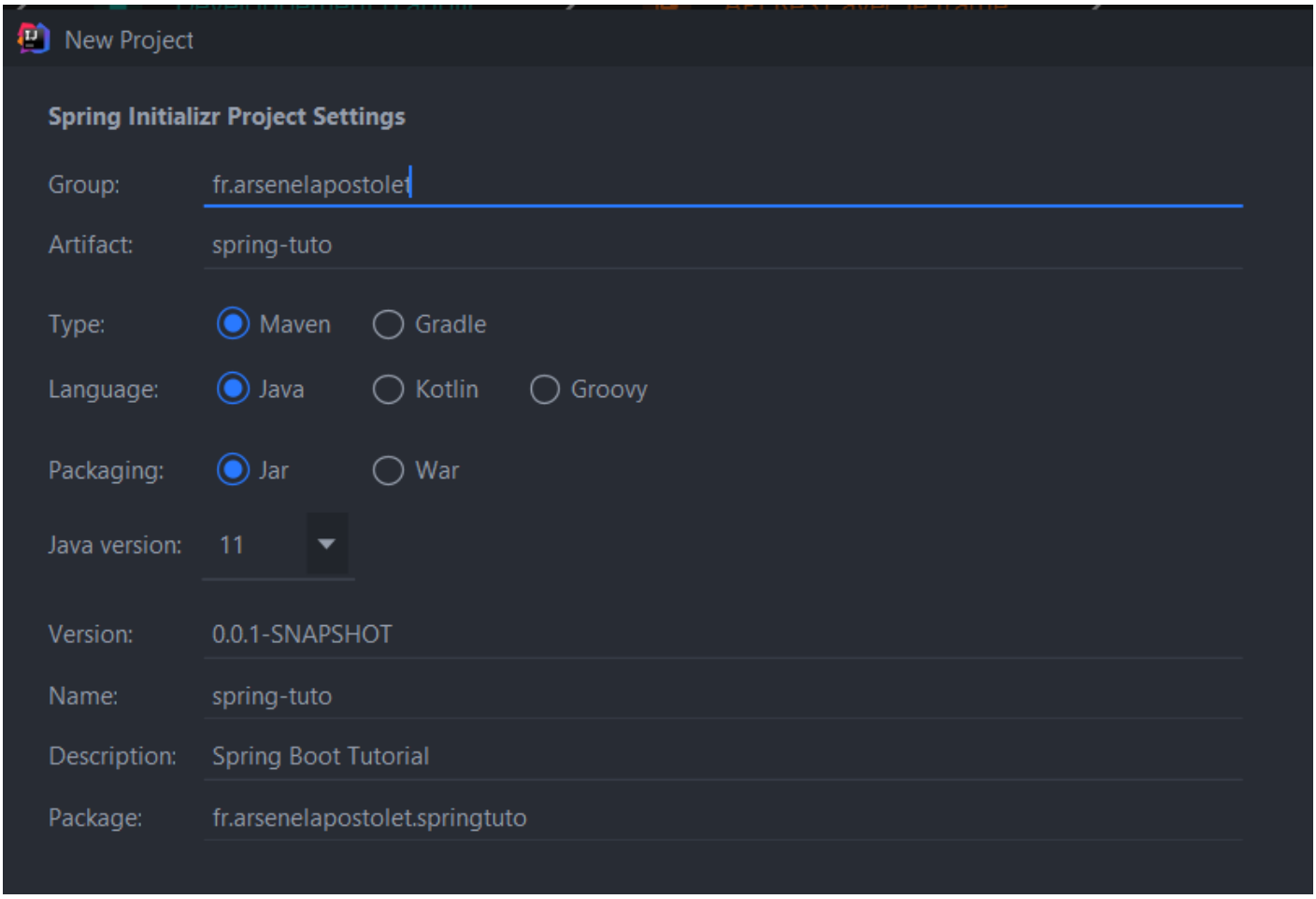
# Créer un projet

Pour créer un nouveau projet avec Spring Boot, choisissez dans IntelliJ le template de projet "Spring Initializer" :

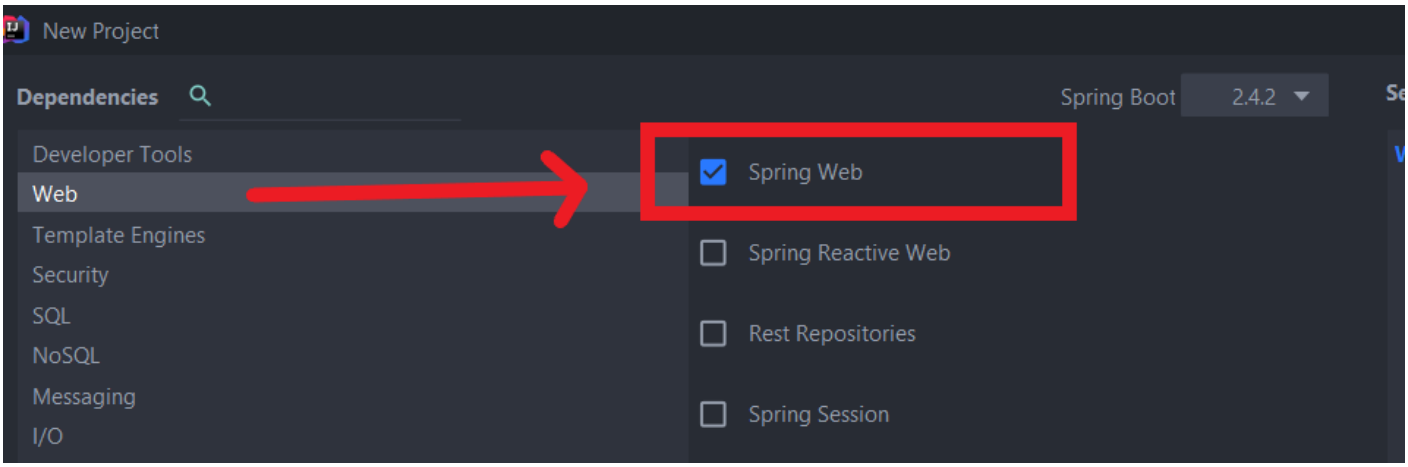



> Si vous êtes sur IntelliJ Community Edition, il faut installer le plugin Spring Initilizer ou alors utiliser le site web.

Choisissez votre SDK Java et faites "Next". Il faut ensuite configurer votre artéfact : son nom, son groupe, le langage et le build tool utilisé :




Nommez votre artéfact, puis votre groupe. Choisissez Java pour le langage et Maven pour le build tool. Une fois que c'est fait, cliquez sur "Next". Enfin, il faut choisir les dépendances de notre projet. Choisissez pour l'instant uniquement "Spring Web" dans la rubrique "Web" :




Voilà votre projet est créé !

# L'API

L'API que nous allons développer au fur et à mesure de ce cours est une API simple de gestion d'une Todo liste. Elle va se baser sur la ressource Todo suivante :

```
public class Todo {

  private String id;
  private String title;
  private String description;

  ... getters & setters ...
}
```

# Bases du développement d'API

## Controleurs

Afin de répondre à des requêtes HTTP, on utilise des controleurs. Ce sont des classes qui vont contenir des méthodes particulières, les méthodes endpoints. Une méthodes endpoint est une méthode qui gère des requêtes HTTP pour une route et une méthode HTTP donnée. Un controleur peut possèder un préfixe de route qui sera le début de la route gérée par ses méthodes endpoints.

Pour développer une API ReST (par opposition avec une application à vues), nous allons donc utiliser l'annotations `@RestController` sur nos controleurs pour les enregistrer auprès du framework.

Commencez par créer un package `controller` qui contiendra tous nos controleurs. Créez ensuite une nouvelle classe, votre premier controleur :

```
@RestController
@RequestMapping("/api/todos")
public class TodoController {


}
```

Ce controleur est notre controleur de Todos, grâce à l'annotation `@RequestMapping` on déclare qu'il va gérer les requêtes sur les routes qui commencent par `/api/todos`.

## Méthode Endpoint

Pour déclarer une méthode Endpoint, il suffit de l'annoter avec `@GetMapping()`, `@PostMapping()`, `@PutMapping()` ... en fonction de la méthode a gérer. La route gérée par la méthode est passée en paramètre de cette annotation. S'il n'est pas renseigné, alors la méthode gère la route racine du controleur pour cette méthode HTTP.

Par défaut dans Spring Boot, lorsque vous retournez un objet d'une méthode endpoint, ce dernier est sérialisé en JSON et le résultat est écrit dans la réponse de la requête.

Exemple :

```
@GetMapping
public List<Todo> getTodos(){
  return Arrays.asList(
    new Todo("3f13bb4c-6d88-4cc5-97c8-868569ac2e94","todo 1","todo 1 description"),
    new Todo("e23d5839-1299-4334-ba35-2a9c62ef17a3","todo 2","todo 2 description")
    );
}
```

Cette méthode Endpoint n'as pas de route précisée, elle va donc gérer la route racine du constructeur pour la méthode GET. Elle retourne également une liste de Todo, cette dernière va être automatique transformée en JSON, le résultat de la requête sera donc :

```
[
  {
    "id":"3f13bb4c-6d88-4cc5-97c8-868569ac2e94",
    "name":"todo 1",
    "description":"todo 1 description"
  },
  {
    "id":"e23d5839-1299-4334-ba35-2a9c62ef17a3",
    "name":"todo 2",
    "description":"todo 2 description"
  }
]
```

# Paramètre d'URL

Une méthode endpoint peut récupérer un paramètre d'URL de la requête grâce à l'annotation `@RequestParameter` qui prend en paramètre le nom du paramètre :

```
@GetMapping
public List<Todo> getTodos(@RequestParam("name") String todoName){
  ...
}
```

Cette méthode récupère en paramètre `todoName` le paramètre d'URL name de la requête.

# Paramètre de route

Une méthode endpoint peut récupérer un paramètre dans le chemin de la requête. Le nom du paramètre est template dans la route de méthode avec des accolades. Il est ensuite récupéré par la méthode grâce à l'annotation `@PathVariable` qui prend en paramètre le nom templaté dans la route :

```
@GetMapping("/{id}")
public Todo getTodoFromId(@PathVariable("id") String id){
...
}
```

Cette méthode ne gère plus les requêtes GET sur `/api/todos` mais sur `/api/todos/quelquechose`. La méthode va récupérer ce "quelquechose" en paramètre.

# Body de la requête

Les méthodes Endpoint qui gère des requêtes dont la méthode HTTP peut contenir un Body peuvent récupérer ce Body sous la forme d'un objet Java avec l'annotation `@RequestBody` :

```
@PostMapping()
public Todo createTodo(@RequestBody Todo todo){
...
}
```

Ici, lors d'une requête POST sur `/api/todos`, la Framework va essayer de désérialiser le Body de la requête (au format JSON) dans un objet Java de type `Todo`.

# Les services

Comme nous l'avons vu dans l'introduction, le framework Spring utilise massivement le principe d'injection de dépendances.

# Injection de dépendance

L'injection de dépendances consiste à une classe instanciée par le framework (comme par exemple nos controlleur), de se faire fournir les classes dont elle dépend par un partie du framework qui s'appelle le conteneur d'injection de dépendance (aussi appelée conteneur d'inversion de controle). Cela peremet donc de découpler intégralement une classe de ses dépendances grâce à la programmation par interface.

# Component Scan

L'injection de dépendance dans Spring Boot utilise un mécanisme appelé le Component Scan, qui permet au conteneur d'injection de dépendance de détecter automatiquement les classes à injecter grâce à des annotation. La principale est l'annotation `@Component` mais elle possède des alias sémantique (ils font la même chose mais permettent de donner plus de sens) comme par exemple `@Service`. Nous allons donc principalement utiliser `@Service`.

# Pourquoi les services ?

Le but des services est de séparer la logique propre à l'application de la logique HTTP (qui réside dans les controleurs), les controleurs ne doivent avoir pour responsabilité que de gérer des requêtes et réponses HTTP et gérer les erreurs proprement. Pour tout traitement logique, nous allons utiliser un service. Le service va prendre en entrée les données traitées par le controleur, effectuer les traitement logique, et si besoin retourner une réponse au controleur.

# Premier service

Pour créer un service il faut d'abord définir son contrat de service sous la forme d'une interface :

```
public interface TodoService {

  ...

}
```

Ensuite, il faut fournir un implémentation de cet interface :

```
@Service
public class TodoServiceImpl implements TodoService {

  ...

}
```

On utilise l'annotation `@Service` pour signaler au conteneur d'injection de dépendance qu'il s'agit d'une classe à scanner. Enfin, notre controleur pourra déclarer ce service comme dépendance en le prennant en paramètre de son fonstructeur :

```

public class TodoController {

  private final TodoService todoService;

  public TodoController(TodoService todoService) {
    this.todoService = todoService;
  }

}
```

Ainsi, leur de la construction du controleur par le framework, notre implémentation de `TodoService` lui sera fournie par le conteneur d'injection de dépendance. De cette façon, le controleur pour utiliser les service établis par le contrat de service définit par `TodoService` sans dépendre d'aucune façon de son implémentation.

# Spring Data JPA

Dans une application backend, on a (presque) toujours besoin de persister des données dans un système de gestion de base de donnée. On pourrait écrire des requêtes JDBC pour tout mais c'est du code super long et répétitif. Pour pallier ce problème, il existe JPA (Java Persistance API), une spécification d'ORM pour Java. Un ORM (Object Relationnal Mapper) est un composant logiciel qui va traduire automatiquement des instances d'objets du modèle orienté objet, en ligne d'enregistrement dans le modèle relationnel. JPA étant une spécification, nous avons besoin d'une implémentation, nous allons utiliser la plus connue qui s'appelle Hibernate.

# Installer les dépendances

Pour installer JPA et Hibernate, rien de plus simple, il suffit d'ajouter la dépendance suivante à notre fichier `pom.xml` :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Il nous faut aussi un Driver JDBC, car ce dernier sera utilisé par Hibernate sous le capot.

Selon votre SGBD :

# MySQL

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.22</version>
</dependency>
```

# Postgres

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
</dependency>
```

## Oracle

```
<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>21.1.0.0</version>
</dependency>
```

# Configurer la connexion

Afin de configurer la connexion à la base de donnée, ouvrez le fichier `application.properties` situé dans `src/main/resources`. Et ajoutez les entrées suivantes :

```
spring.datasource.url=ici l'url de ma base
spring.datasource.username=ici le login de ma base
spring.datasource.password=ici le mot de passe de ma base

spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.temp.use_jdbc_metadata_defaults=false
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

# Les Entités

Les entités sont les objets qui sont persistés par l'ORM dans la base de donnée. Pour définir nos classes d'objets qui seront enregistrées dans la base de donnée, il faut les définir en tant qu'entité. Cela se fait à l'aide de l'annotation `@Entity` sur la classe. Une entité doit avoir un champs qui correspondra à se clé primaire dans la base de donnée, il est désigné avec l'annotation `@Id`. l'ORM peut le générer automatiquement, et même aléatoirement dans le cas d'une UUID (voir exemple).

Les autres champs de la classe entités sont persistés automatiquement. Pour qu'un champs ne soit pas persisté, il faut l'annoter avec `@Transient`.

Exemple d'entité pour notre Todo :

```
@Entity
public class Todo {
  @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

  private String title;
    private String description;

  ... getters & setters ...
}
```

Les annotations `@GeneratedValue` et `GenericGenerator` permet de générer automatiquement un UUID aléatoire pour l'entité.

On peut aussi définir des contraintes sur les colonnes en rajoutant l'annotation `@Column` sur le champs et en lui passant des paramètres, par exemple :

```
@Column(unique = true, length = 32)
```

Cela permet de mettre la contrainte "unique" sur le champs et d'imposer une longueur maximum de 32 caractère

# Les Repository

Pour interagir avec les entités, il faut créer des Repository. Ce sont des interface que nous allons définir, mais qui seront implémentées non pas nous, mais pas l'ORM Hibernate. Ils peuvent ensuite être injectés dans les services par le conteneur d'injection de dépendances. Une interface Repository gère les interactions pour une entité et doit étendre l'interface `JpaRepository` en fournissant en paramètre de type, le type de l'entité, ainsi que le type de son Id.

Exemple :

```
public interface TodoRepository extends JpaRepository<Todo, String> {


}
```

# Méthodes de base

Voilà, juste en étendant cette interface, on peut accèder à tout un tas de méthodes intéressantes. Les principales sont :

- `save` : Sauvegarder (créer ou mettre à jours) une instance d'une entités
- `findById` : récupérer une entité à partir de son Id
- `findAll` : récupérer toutes les entités contenues dans la base (⚠ attention ça peut faire beaucoup de tout charger dans la mémoire)
- `deletebyId` : supprimer une entité à partir de son Id

Et bien d'autre qui peuvent être utile, à retrouver dans la javadoc de l'interface

# Méthodes Custom

On peut également définir de nouvelles méthodes dans l'interface pour filtrer sur d'autre champs, en nommant ces méthode de façon particulière :

```
findByTitle(String title);
```

`findBy` veut dire qu'on filtre sur un champs et `Title` est le nom d'un champs de notre entité, cette requête va donc récupérer les Todos qui ont le titre passé en paramètre.

Voici une références des mots utilisables dans les noms de méthode

| Mot | Exemple | Logique SQL correspondante |
|---|---|---|
| And | findByLastnameAndFirstname | ... where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | ... where x.lastname = ?1 or x.firstname = ?2 |
| Is, Equals | findByFirstname,findByFirstnameIs,findByFirstnameEquals | ... where x.firstname = ?1 |
| Between | findByStartDateBetween | ... where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | ... where x.age < ?1 |

| Mot | Exemple | Logique SQL correspondante |
|---|---|---|
| LessThanEqual | findByAgeLessThanEqual | ... where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | ... where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | ... where x.age >= ?1 |
| After | findByStartDateAfter | ... where x.startDate > ?1 |
| Before | findByStartDateBefore | ... where x.startDate < ?1 |
| IsNull, Null | findByAge(Is)Null | ... where x.age is null |
| IsNotNull, NotNull | findByAge(Is)NotNull | ... where x.age not null |
| Like | findByFirstnameLike | ... where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | ... where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | ... where x.firstname like ?1 (parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | ... where x.firstname like ?1 (parameter bound with prepended %) |
| Containing | findByFirstnameContaining | ... where x.firstname like ?1 (parameter bound wrapped in %) |
| OrderBy | findByAgeOrderByLastnameDesc | ... where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | ... where x.lastname <> ?1 |
| In | findByAgeIn(Collection ages) | ... where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection ages) | ... where x.age not in ?1 |
| True | findByActiveTrue() | ... where x.active = true |
| False | findByActiveFalse() | ... where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | ... where UPPER(x.firstname) = UPPER(?1) |

# Relations

Les relations permettent de faire des liens entre les entités. Il existe quatres types de relations, définies par des annotations :

- `@ManyToOne` : plusieurs instances de cette classe entité sont en relation avec une unique instance d'une autre (ex: Pages d'un livre - plusieurs pages sont reliées à un unique livre)
- `OneToMany` : une unique instance de cette classe entité est en relations avec plusieurs instances d'une autre (ex: Livre qui contient des pages - un unique livre est relié à plusieurs pages)

- `@ManyToMany` : plusieurs instance de cette classe entité sont en relations avec plusieurs isntance d'une autre (ex: Classes et Professeurs - Les professeurs ont plusieurs classes et les classes ont plusieurs professeurs.
- `@OneToOne` : une unique instance de cette classe entités est reliée à une unique instance d'une autre (ex: Dircteur et Ecole - une directeur dirige une seule école et une école est dirrigée par un seul directeur).

Exemple :

```
@Entity
public class Todo {
  ... autres propriétés ...

  @ManyToOne
  private ApplicationUser owner;

  ... getters & setters ....
}
```

```
@Entity
public class ApplicationUser {
    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

  private String name;

  @OneToMany
    private Set<Todo> todos;

    ... getters & setters ...
}
```

Les annotations de relation possèdes plusieurs paramètres utiles à connaitres :

- `fetch` :
  - `FetchType.EAGER` : charge les données de la relation directement (défaut pour `@ManyToOne` et `@OneToOne`)
  - `FtechType.LAZY` : charge les données de la relation que quand le getter est appelé (défaut pour `@OneToMany` et `@ManyToMany`

- `orphanRemoval` : `true` ou `false`, détermine dans un `@OneToMany` si les enfant doivent être supprimés quand le parent est supprimé → un enfant ne peut exister sans parent.

# Le Pattern DTO

Quand une API dépasse le simple CRUD sur une entité, on commence à avoir besoin formaliser les entrées et les sorties de notre API. Par exemple losqu'on a des entités JPA et qu'on ne veut pas les exposer intergralement au niveau du client pour différentes raisons. Par exemple, si on essaye de sérialiser en JSON une entité membre d'une relation à double navigation (le parent voit ses enfants et les enfants voient le parent) on aura une boucle finie.

# Les Data Transfer Objects

Les DTO sont des classes toutes simples qui ne contiennent que des propriétés (champs privé avec getter & setters) et sont utilisées comme paramètre et valeur de retour des controleurs. Ils voyagent également jusque dans les services une fois validés. On va donc avoir tendance à créer pour chaque forme de requête et de réponse un nouveau DTO.

Exemple :

```
public class CreateTodoDTO {


  private String title;
  private String description;


  ... getters & setters ...
}
```

Pour créer un Todo, l'Id n'est pas spécifié par l'utilisateur, on va donc créer un DTO de Todo sans Id, et le prendre en paramètre lors de la création des Todos.

Aussi :

```
public class TodoDTO {
  private String id;
  private String title;
  private String description;
  private String ownerId;
  ... getters & setters ...
}
```

Pour notre DTO qui permet d'envoyer un Todo au client, on rajoute l'Id, et on remplace la référence à l'utilisateur par uniquement son Id, afin d'éviter la boucle infinie étant donné que la référence à l'user liste elle même les Todos.

# Validation Automatique

Spring Boot implémente un standard de validation Java (javax.validation) des DTO basé sur des annotations, qui permet de façon déclarative d'appliquer des contraintes sur les propriétés des DTO.

Installation :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Les principales annotations disponibles :

- `@NotNull` : pas de valeur `null`
- `@NotEmpty` : pour les `String` et les `Collection`, la taille doit être supérieure à 0
- `@NotBlank` : pour les `String`, imposer que la chaine ne doit pas être vide
- `@AssertTrue` / `@AssertFalse` : pour les `boolean`
- `@Min` / `@Max` : pour les valeur numérques, pour imposer in minimum / maximum
- `@Positive` / `@PositiveOrZero` / `@Negative` / `@NegativeOrZero` : permet de contraindres les valeurs numériques selon le signe
- `@Email` : pour vérifier qu'une `String` a la forme d'une email
- `@Future` / `@Past` / `@FuturOrPresent` / `@PastOrPresent` : pour contraindre les dates par rapport à la date courante

Les annotations peuvent aussi être utilisées sur les objets dans les collections : `List<@NotBlank String>`. En utilisant ces annotations, les DTO sont automatiquement validés par le framework lorsqu'ils passent pas le controleur lorsqu'il sont marqués par l'annotation `@Valid`.

Exemple pour nos DTO de Todos :

```
public class CreateTodoDTO {
  @Size(min=6, max=255)
  private String title;

  @NotEmpty
  private String description;
```

```
  ... getters & setters ...
}
```

Et dans notre méthode endpoitn :

```
@PostMapping
public void createTodo(@Valid @RequestBody CreateTodoDTO dto){
...
}
```

# Gérer les codes de réponse HTTP

Dans le standard REST, les codes de retours HTTP sont importants car ils ont une sémantique. Il convient donc de retourner les bons codes de réponse HTTP dans chacun de nos endpoints.

# Cas nominal

Pour le cas nominal, on peut utilise l'annotation `@ResponseStatus` sur la méthode endpoint. On lui passe en paramètre le status à l'aide de l'annotation `HttpStatus`.

Exemple :

```
@GetMapping
@ResponseStatus(HttpStatus.CREATED)
public TodoDTO createTodo(@RequestBody CreateTodoDTO dto){
...
}
```

# Cas d'erreur

Pour gérer les cas d'erreur on peut lancer des exceptions spécifiques définies par le framework, comme par exemple :

- `BadRequestException` : pour le status 400 Bad Request
- `ResourceNotFoundException` : pour le status 404 not found.

Il est également possible d'attribuer des codes de retours à des exception personnalisées qui étende `RuntimeException` avec l'annotation `@ResponseStatus` :

```
@ResponseStatus(value = HttpStatus.I_AM_A_TEAPOT)
public class IamATeapotException extends RuntimeException {
    public MyResourceNotFoundException() {
```

```
        super();
    }
    public MyResourceNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    public MyResourceNotFoundException(String message) {
        super(message);
    }
    public MyResourceNotFoundException(Throwable cause) {
        super(cause);
    }
}
```

# Sécuriser l'API avec JWT

JWT (JSON Web Token) est un standard de sécurité qui a pour but l'authentification et l'autorisation des clients. La particularité de JWT est qu'il permet de générer un token qui non seulement authentifie le client mais contient aussi des informations signées (on peut en vérifier l'intégraté) à propos de l'utilisateur. Il s'agit aussi d'une authentification sans état (contrairement aux cookies ou à la session coté serveur), ce qui permet de respecter le caractère sans état du standard ReST. Pour sécuriser notre API, nous allons utiliser Spring Security, et nous allons y intégrer JSON Web Tokens.

# Installation des dépendances

## Spring Security

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

## JSON Web Token

```
<dependency>
  <groupId>com.auth0</groupId>
  <artifactId>java-jwt</artifactId>
  <version>3.11.0</version>
</dependency>
```

Ensuite rajoutez le code suivante dans votre `SpringBootTodoApplication` :

```
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
  return new BCryptPasswordEncoder();
}
```

Cela permet d'importer un algorithme de hachage (BCrypt) pour hacher les motes de passe des utilisateur dans le stockage.

Créez ensuite un package `security` afin de mettre tout notre code de configuration de sécurité.

# Constantes

Pour commencer, on va créer une classe `SecurityConstantes` dans laquelle nous allons stocker toutes les valeurs dont nous auront besoin :

```
public class SecurityConstants {
    public static final String SECRET = "SecretKeyToGenJWTs";
    public static final long EXPIRATION_TIME = 864_000_000; // 10 days
    public static final String TOKEN_PREFIX = "Bearer ";
    public static final String HEADER_STRING = "Authorization";
    public static final String SIGN_UP_URL = "/api/users/sign-up";
}
```

- `SECRET` : secret utilisé pour l'algorithme cryptographique de signature du token JWT, doit être le plus random possible.
- `EXPIRATION_TIME` : temps de validité du token JWT
- `TOKEN_PREFIX` : préfixe du token JWT dans le header des requêtes. On utilise `Bearer` par convention
- `HEADER_STRING` : nom du header des requêtes utilisé pour passer le token. On utilise `Authorization` par convention
- `SIGN_UP_URL` : URL pour l'inscription des nouveaux utilisateurs

# Authentification

Créez une nouvelle classe `JWTAuthenticationFilter` afin de configurer un filtre d'authentification, c'est à dire vérifier que l'utilisateur a les bons credentials pour lui délivrer un token.

Cette classe va étendre `UsernamePasswordAuthenticationFilter`. Il faut également prendre en paramètre un `AuthenticationManager` (fourni pas le conteneur d'injection de dépendances) afin de pouvoir interagir avec le framework de sécurité. Enfin, on redéfinit les deux méthodes qui nous intéressent : `attemptAuthentication` et `successfulAuthentication`. La première va s'occuper de vérifier les credentials de l'user et la deuxième va générer son token.

```
public class JWTAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
    private AuthenticationManager authenticationManager;
```

```
  public JWTAuthenticationFilter(AuthenticationManager authenticationManager) {
     this.authenticationManager = authenticationManager;
  }

 @Override
  public Authentication attemptAuthentication(HttpServletRequest req,
                                      HttpServletResponse res) throws AuthenticationException {

   }

 @Override
  protected void successfulAuthentication(HttpServletRequest req,
                                      HttpServletResponse res,
                                      FilterChain chain,
                                      Authentication auth) throws IOException, ServletException {

   }
```

Dans la première méthode, on va récupérer les credentials de l'utilisateur dans la requête, puis on les passe au framework :

```
  @Override
  public Authentication attemptAuthentication(HttpServletRequest req,
                                      HttpServletResponse res) throws AuthenticationException {
     try {
        ApplicationUser creds = new ObjectMapper()
              .readValue(req.getInputStream(), ApplicationUser.class);

        return authenticationManager.authenticate(
              new UsernamePasswordAuthenticationToken(
                    creds.getUsername(),
                    creds.getPassword(),
                    new ArrayList<>())
        );
     } catch (IOException e) {
        throw new RuntimeException(e);
     }
  }
```

Ce filtre va permettre de gérer les requête POST sur `/login` contenant un `ApplicationUser` dans le Body de la requête.

Dans la seconde, on crée le token de l'utilisateur, et on le place dans le header prévu de la réponse :

```
    @Override
    protected void successfulAuthentication(HttpServletRequest req,
                                            HttpServletResponse res,
                                            FilterChain chain,
                                            Authentication auth) throws IOException, ServletException {


        String token = JWT.create()
                .withSubject(((User) auth.getPrincipal()).getUsername())
                .withExpiresAt(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
                .sign(HMAC512(SECRET.getBytes()));
        res.addHeader("Access-Control-Expose-Headers", "Authorization");
        res.addHeader("Access-Control-Allow-Headers", "Authorization, X-PINGOTHER, Origin, X-Requested-With,
Content-Type, Accept, X-Custom-header");
        res.addHeader(SecurityConstants.HEADER_STRING, SecurityConstants.TOKEN_PREFIX + token)
    }
}
```

On a aussi besoin d'ajouter quelques header pour dire à la sécurité du navigateur que le client a le droit de lire le header du token.

# Autorisation

Créez une nouvelle classe `JWTAuthenticationFilter` afin de configurer un filtre d'autorisation, c'est à dire vérifier que l'utilisateur qui cherche à accèder une ressource protégée, prossède bien un token valide.

Cette classe va étendre `BasicAuthenticationFilter`. Il faut également prendre en paramètre un `AuthenticationManager` (fourni pas le conteneur d'injection de dépendances) afin de pouvoir interagir avec le framework de sécurité. Enfin, on redéfinit une qui nous intéresse : `doFilterInternal`. Elle va dire au framework si l'utilisateur est bien authentifié ou non. Pour éviter que cette méthode soit trop longue nous allons créer une méthode `getAuthentication` qui va s'occuper de valider le token.

```
public class JWTAuthorizationFilter extends BasicAuthenticationFilter {
```

```
    public JWTAuthorizationFilter(AuthenticationManager authManager) {
        super(authManager);
    }

    @Override
    protected void doFilterInternal(HttpServletRequest req,
                                    HttpServletResponse res,
                                    FilterChain chain) throws IOException, ServletException {
□□...
    }

    private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {
□□...
    }
}
```

Dans la première méthode on va récupérer le token dans la requête et le passer à la seconde méthode :

```
@Override
protected void doFilterInternal(HttpServletRequest req,
                                HttpServletResponse res,
                                FilterChain chain) throws IOException, ServletException {
  String header = req.getHeader(HEADER_STRING);

  if (header == null || !header.startsWith(TOKEN_PREFIX)) {
    chain.doFilter(req, res);
    return;
  }

  UsernamePasswordAuthenticationToken authentication = getAuthentication(req);

  SecurityContextHolder.getContext().setAuthentication(authentication);
  chain.doFilter(req, res);
}
```

Ensuite on implémente `getAuthentication` pour valider le token :

```
private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {
  String token = request.getHeader(HEADER_STRING);
```

```
  if (token != null) {

    String user = JWT.require(Algorithm.HMAC512(SECRET.getBytes()))
      .build()
      .verify(token.replace(TOKEN_PREFIX, ""))
      .getSubject();

    if (user != null) {
      return new UsernamePasswordAuthenticationToken(user, null, new ArrayList<>());
    }
    return null;
  }
  return null;
}
```

# Récupérer les utilisateurs

Maintenant, pour que l'authentification puisse marcher, il faut indiquer au framework comment trouver nos utilisateurs. On va donc créer une classe `UserDetailsServiceImpl` qui étend `UserDetailsService`.

Nous allons lui injecter notre repository JPA qui permet de retrouver les utilisateur, et l'utilisateur pour envoyer notre utilisateur au framework :

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {
    private ApplicationUserRepository applicationUserRepository;

    public UserDetailsServiceImpl(ApplicationUserRepository applicationUserRepository) {
        this.applicationUserRepository = applicationUserRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        ApplicationUser applicationUser = applicationUserRepository.findByUsername(username);
        if (applicationUser == null) {
            throw new UsernameNotFoundException(username);
        }
        return new User(applicationUser.getUsername(), applicationUser.getPassword(), emptyList());
```

```
    }
}
```

Si l'utilisateur n'existe pas, on lance une `UsernameNotFoundException`.

# Configuration de Spring Security

On va maintenant créer une classe `WebSecurity` qui va nous permettre de configurer la sécurité en faisant le lien entre tous les composants que nous avons créés ainsi que le framework.

Nous allons lui injecter notre algo de vérification des mots de passe, `BCrypt` ainsi que notre service de récupération d'utilisateur. Nous allons aussi redéfinir deux méthodes `configure` afin de relier tous les éléments dont nous avons besoin :

```
@EnableWebSecurity
public class WebSecurity extends WebSecurityConfigurerAdapter {
    private UserDetailsServiceImpl userDetailsService;
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public WebSecurity(UserDetailsServiceImpl userDetailsService, BCryptPasswordEncoder
bCryptPasswordEncoder) {
        this.userDetailsService = userDetailsService;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
□□...
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        ...
    }
}
```

Pour la première méthode, nous allons configurer quelles routes sont protégées. On va définir celles qui ne seront pas protégé comme cas spécial, et toutes les autres seront protégées. Ici on a définit que l'URL d'inscription n'étais pas protégée.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.cors().and().csrf().disable().authorizeRequests()
    .antMatchers(HttpMethod.POST, SIGN_UP_URL).permitAll()
    .anyRequest().authenticated()
    .and()
    .addFilter(new JWTAuthenticationFilter(authenticationManager()))
    .addFilter(new JWTAuthorizationFilter(authenticationManager()))
    .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}
```

Pour rajouter des routes non protégées, il faut rajouter une ligne avec la méthode et l'url de la route :

```
.antMatchers(HttpMethod.POST, "mon url").permitAll()
```

Et ce juste avant `.anyRequest().authenticated()`. Ensuite nous ajoutons nos filtres and nous désactivons les session car avec JWT il n'y en a pas besoin.

Enfin, la seconde méthode `configure` sert juste à passer notre service de récupération d'utilisateurs au framework :

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
  auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder);
}
```

Voilà ! Notre récurité est configurée, il ne nous reste plus qu'à implémenter l'inscription des utilisateurs.

# Inscription

Pour l'inscritpion il suffit de créer un endpoint qui permet d'insérer un nouvel utilisateur dans la base en utilisant la route que nous avons préparée pour ce faire dans la configuration de sécurité (POST `/api/users/sign-up`), sans oublier de hacher son mot de passe avec Bcrypt, et le tour est joué !

# Packager une application client riche avec l'API

Nous allons apprendre comment packager une application frontend pour la compiler puis la servir avec notre API sur la route `/`. Nous allons prendre l'exemple d'une application Angular.

Tout d'abord, créez un dossier `frontend` dans `src/main/resources` et créez ou déplacez votre projet angular dedans. Ensuite dans votre fichier `angular.json` (situé à la racine de votre projet angular), cherchez `outputPath` et affectez y la valeur `../public`.

Cela va permettre de dire à votre projet Angular de stocker son résultat de compilation dans le dossier des fichiers statiquement servis par Spring Boot.

Ensuite, afin de compiler le projet Angular avec Maven, rajouter dans votre `pom.xml` le plugin suivant :

```
<plugin>
            <groupId>com.github.eirslett</groupId>
            <artifactId>frontend-maven-plugin</artifactId>
            <version>1.6</version>
            <configuration>
               <workingDirectory>src/main/resources/frontend</workingDirectory>
               <!-- where to install npm -->
               <installDirectory>${project.build.directory}/install</installDirectory>
            </configuration>
            <executions>
               <execution>
                  <id>install-node-and-npm</id>
                  <goals>
                     <goal>install-node-and-npm</goal>
                  </goals>
                  <configuration>
                     <nodeVersion>latest-v14.x/</nodeVersion>
                     <npmVersion>6.14.10</npmVersion>
                  </configuration>
               </execution>
               <execution>
```

```
            <id>npm-install</id>
            <goals>
              <goal>npm</goal>
            </goals>
            <configuration>
              <arguments>install</arguments>
            </configuration>
          </execution>
          <execution>
            <id>build</id>
            <goals>
              <goal>npm</goal>
            </goals>
            <configuration>
              <arguments>run build</arguments>
            </configuration>
          </execution>
        </executions>
      </plugin>
```

Voilà ! Maintenant, en tappant :

```
mvn clean package
```

Le projet Angular sera généré et packagé avec le projet Spring, qui servira statiquement le front sur la route `/`. Cela prend un peu de temps la première fois à cause de l'installer de NodeJS, de NPM et l'exécution de `npm install` qui téléchargem toutes les dépendances.

# Swagger UI

Swagger UI est un outil de documentation automatique qui va générer, à partir de votre code, une page de documentation interactive.

# Installation des dépendances

Ajoutez cette dépendance à votre `pom.xml` :

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.5.2</version>
</dependency>
```

Et rafraichissez vos dépendances maven.

# Configuration

# Tests d'intégration

# Intégration continue avec Github Actions

# Déploiement continu sur Heroku